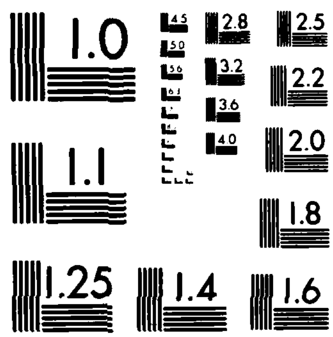
AD-A151 615 SISAL OPTIMIZATIONS(U) AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OH J D ACRES 1984 AFIT/CI/NR-85-13T 1/1

UNCLASSIFIED F/G 9/2 NL

END

FILMED

DTIC



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

UNCLASS

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: AFIT/CI/NR 85-13T
2. GOVT ACCESSION NO.:
3. RECIPIENT'S CATALOG NUMBER:
4. TITLE (and Subtitle): Sisal Optimizations
5. TYPE OF REPORT & PERIOD COVERED: THESIS/D~~ISSERTATION~~
6. PERFORMING ORG. REPORT NUMBER:
7. AUTHOR(s): Jody Dejonghe Acres
8. CONTRACT OR GRANT NUMBER(s):
9. PERFORMING ORGANIZATION NAME AND ADDRESS: AFIT STUDENT AT: Colorado State Univ
10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS:
11. CONTROLLING OFFICE NAME AND ADDRESS: AFIT/NR WPAFB OH 45433
12. REPORT DATE: 1984
13. NUMBER OF PAGES: 36
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office):
15. SECURITY CLASS. (of this report): UNCLASS
15a. DECLASSIFICATION/DOWNGRADING SCHEDULE
16. DISTRIBUTION STATEMENT (of this Report): APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)
18. SUPPLEMENTARY NOTES: APPROVED FOR PUBLIC RELEASE: IAW AFR 190-1

LYNN E. WOLAVER 21 Feb 85
Dean for Research and Professional Development
AFIT, Wright-Patterson AFB OH

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)
20. ABSTRACT (Continue on reverse side if necessary and identify by block number): ATTACHED

DTIC ELECTE MAR 25 1985 S D

D

AD-A151 615

DTIC FILE COPY

DD FORM 1473 1 JAN 73 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASS

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

85 03 11 066

ABSTRACT

The goal of this master's project was to provide several optimizations for the SISAL compiler being developed at Colorado State University. SISAL is a data flow language intended for use on a variety of multiprocessor architectures. Since SISAL is compiled into the intermediate form IF1, which is a common intermediate form for data flow languages, this project concentrated on optimizations that are unique to the characteristics of SISAL, rather than on more traditional optimizations. In particular, the optimizations developed concentrated on making array operations more efficient by deleting operations where possible and by performing "live analysis" which would tell if data values will ever be needed again in a program.

AFIT RESEARCH ASSESSMENT

The purpose of this questionnaire is to ascertain the value and/or contribution of research accomplished by students or faculty of the Air Force Institute of Technology (AU). It would be greatly appreciated if you would complete the following questionnaire and return it to:

AFIT/NR
Wright-Patterson AFB OH 45433

RESEARCH TITLE: Sisal Optimizations

AUTHOR: ACRES, Jody Dejonghe

RESEARCH ASSESSMENT QUESTIONS:

1. Did this research contribute to a current Air Force project?

( ) a. YES ( ) b. NO

2. Do you believe this research topic is significant enough that it would have been researched (or contracted) by your organization or another agency if AFIT had not?

( ) a. YES ( ) b. NO

3. The benefits of AFIT research can often be expressed by the equivalent value that your agency achieved/received by virtue of AFIT performing the research. Can you estimate what this research would have cost if it had been accomplished under contract or if it had been done in-house in terms of manpower and/or dollars?

( ) a. MAN-YEARS ________ ( ) b. $________

4. Often it is not possible to attach equivalent dollar values to research, although the results of the research may, in fact, be important. Whether or not you were able to establish an equivalent value for this research (3. above), what is your estimate of its significance?

( ) a. HIGHLY SIGNIFICANT ( ) b. SIGNIFICANT ( ) c. SLIGHTLY SIGNIFICANT ( ) d. OF NO SIGNIFICANCE

5. AFIT welcomes any further comments you may have on the above questions, or any additional details concerning the current application, future potential, or other value of this research. Please use the bottom part of this questionnaire for your statement(s).

NAME GRADE POSITION

ORGANIZATION LOCATION

STATEMENT(s):

S I S A L   O P T I M I Z A T I O N S

JODY DEJONGHE ACRES
CAPT, USAF
FALL 1984

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR
MASTER OF SCIENCE
COMPUTER SCIENCE
COLORADO STATE UNIVERSITY

(36 PAGES)

1.0 OVERVIEW

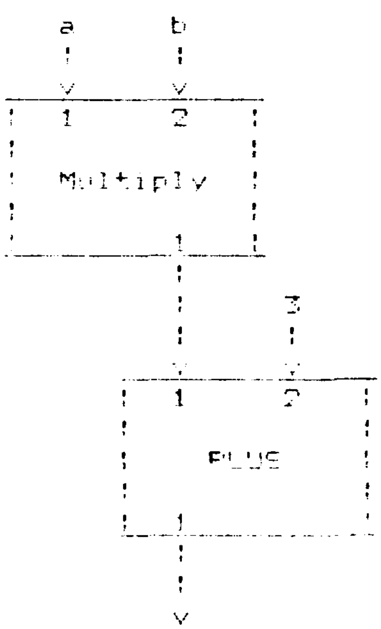
The purpose of this master's project was to provide several optimizations needed for the SISAL compiler being developed at Colorado State University. In particular the optimizations considered were loop invariant removal, array optimizations and stream optimizations. The majority of the standard optimizations applied to compilers will be provided by other sources. This project's main goal, therefore, was to provide "non-standard" optimizations that are required by some of the more unique characteristics of the SISAL compiler.

2.0 BACKGROUND

SISAL (Streams and Iteration in a Single Assignment Language) is a data flow language intended for use on a variety of multiprocessor architectures. The current project at CSU is to produce a compiler for a Denelcor HEP multiprocessor. The basic characteristics of SISAL are: 1) no side effects; 2) locality of effect; 3) parallelism constrained only by data dependencies; and 4) single assignment. [Cobb, 1984] The main objective of SISAL is to provide a programming language for expressing algorithms which will make it easy to detect and exploit any implicit parallelism. The main application area being numerical computations that are straining current high performance machines. Being a "data flow" language, the statements in SISAL are not necessarily executed sequentially. Instead, they are executed whenever their input data is available. The parallelism is obtained since all instructions

whose data is available can theoretically be executed simultaneously. See [McGraw, 1983] for more information on SISAL.

The SISAL compiler produces an intermediate form referred to as "IF1". IF1 is an intermediate form designed specifically for data flow languages. It describes a program as a data flow graph with nodes representing operations, edges representing data and types associated with an edge identifying the characteristics of the data being passed. Figure 1 shows an example of the graphical representation of the expression (a * b) + 3 and the IF1 description of this graph. The numbers in the boxes are port numbers that are used to identify the values. See [Skedzielewski,1984] for a complete description of IF1.




```
T 100  1  3

G 0

N 1 MULTIPLY
E  0 1  1 1  "a"
E  0 2  1 2  "b"

N 2 PLUS
E  1 1  2 1
E  0 0  2 2  "3"

E  2 1  0 1
```

FIGURE 1

To make the IF1 easier to traverse during compilation, it is encoded into a graphical format using pointers. The nodes, ports, and edges are all represented by a "C" structure containing information about each. They are linked together via pointers which allow one to traverse the graph in either a forward or backward direction. To follow the graph forward one goes from a node to a port (representing an output value), to the edge (representing a use of this value), to a node that uses the data. To follow the graph backward, one goes from a node to an edge representing an input value (physically different than the edge followed in the forward direction) to the port that this value came from, back to the node that produced the value. Figure 2 shows this graphical encoding of the previous example (a * 5) + 7. Going forward from a port and following the linked list of edges will give you all uses of this value. Going backward from a node and following this linked list of edges will give you all the input values for this node. The optimizations performed in this project were applied to this graphical encoding of a SISAL program.

```
            ______________
           |              |
           |    Graph     |
           |______________|
             ^          ^
             |          |
             v          |
        [Port 1]---->[Port 2]
         |     ^       ^     |
         v     |       |     v
      {Edge} {Edge}-->{Edge} {Edge}
        |       ^              |
        v       |              v
       ________________________
       |                      |
       |                      |
       |       Multiply       |
       |                      |
       |______________________|
                  ^
                  |
                  v
           [Port 1]-------->[Port 2]
            |     ^         ^      |
            v     |         |      v
         {Edge}  {Edge}-->{Edge} {Edge}
            |       ^              |
            v       |              v
            ________________________
            |                      |
            |                      |
            |         PLUS         |
            |                      |
            |______________________|
                 |
                 |
                 v
```

FIGURE 2

7.0 LOOP OPTIMIZATIONS

The loop optimization considered in this project was loop invariant removal. This involves finding an instruction inside of a loop whose computation stays constant for each iteration of the loop and moving it outside of the loop. This way it is only executed once saving execution time. Because of the characteristics of SISAL and the graphical encoding of IF1, identifying loop invariants is simpler than classical approaches to solving this problem. In particular, basic blocks and loops are already identified in the intermediate form and use-definition information is contained in the graphical encoding. Moving the code outside of the loop, however, is more difficult.

7.1 LOOP INVARIANT REMOVAL ALGORITHM

The steps involved in removing loop invariants from IF1 are:

1) Traverse the graph recursively backwards and for each "forloop" or "forall" node found perform steps 2-6.

2) Traverse the nodes in the body of the loop recursively backwards marking each node as invariant if all of its input values are invariant. An input value is considered invariant if: a) it is a constant; b) its source node is node 0 meaning it comes from outside the loop or c) its source node has been previously marked as invariant. If a loop node is encountered during this traversal, steps 1-6 are performed on this node before it is processed.

3) Once all invariant nodes have been identified, they are

moved immediately before the loop node. This may cause nodes to be renumbered.

4) The input edges to the nodes moved in 3 must be modified to point to the proper source port. This will be unchanged if the source was another moved node or a constant. If the source port was from node 0, however, the corresponding source from outside the loop must be found and the edge wired into it.

5) Any outputs from the nodes moved in 3 that are still needed inside the loop must be channeled into node 0 of the loop and all nodes that use these values must be hooked up to these new ports.

6) Finally, the ports of node 0 may need to be compacted. Since the only users of a particular input value may have been nodes moved outside the loop, some ports can be deleted and the remaining ports compacted.

3.2 STATUS OF LOOP OPTIMIZATIONS

Steps 1-3 identified above had been completed when it was discovered that this process had already been done at another site. We received copies of the code and executable object and after running several tests and analyzing the code I verified that it did indeed remove loop invariants properly. It was decided to stop any further work on this effort and continue working on optimizations that were unique to the work being done at CSU. Some documentation explaining the loop optimization code we received is contained in Appdx A. Although the code executes correctly I did have some questions about the code

itself. First it seems to only consider an input value as invariant if its source node is the graph node (node 0). In our system, however, constants have no source node. I did run some tests with constants and they were optimized correctly, so their encoding must not be different than ours. Also I don't think the code we received was actually executable. The one thing in nesting levels of procedures was not correct. In addition, I couldn't find any place in the code that fixed the inputs to nodes that weren't moved to reflect the correct location of inputs that were moved. There were, however, many routines that were not supplied with the code and this part could have been hidden in one of them.

4.2 ARRAY OPTIMIZATIONS

The next task was to analyze the array processing in SISAL and to determine if there were any optimizations that could be performed to make it more efficient. This involved first [illegible] the interpretation of SISAL into IF1 and then into the runtime routines that actually perform the array actions to see what additional work, if any, was needed.

4.2.1 ARRAY ANALYSIS

This section outlines how each array operation in SISAL is [illegible] to the runtime routine that actually performs the [illegible] operation.

[illegible]

[illegible]

[illegible]

```
IF1    -
              int   value      value
               |     |..........|
               |     |          |
               v     v
    -------------------------------------------
    |                                          |
    |                ABUILD                    |
    |                                          |
    -------------------------------------------
               |
               v
            array(T)
```

[illegible] - create_el

Params:

NAME	MEANING	SOURCE
numdim	number of bounds	count input edges
numval	number of values	count input edges
[illegible]	element size of values	type of input edge
dvflag	boolean indicating if values are arrays	type of input edge
newdvptr	new dope vector pointer	output param
lbarray	array of lower bounds	input edges
vlarray	array of pointers to input values	input edges

Create/Fill - creates an array with lower and upper bounds, LO and HI filled with value V

SISAL - array_fill(LO,HI,V)

```
IF1 -
              low  hi   value
               |    |     |
               v    v     v
         ------------------------
         |                      |
         |        AFILL         |
         |                      |
         ------------------------
               |
               v
            array(T)
```

[illegible] - create_fill

Params:

NAME	MEANING	SOURCE
[illegible]	the lower bound	input edge
[illegible]	the upper bound	input edge
[illegible]	the element [illegible]	type of edge
d[illegible]	boolean indicating if [illegible] array	type of edge
[illegible]	new dope vector	output param
[illegible]	[illegible] the value	input edge

Select - returns a pointer to the array element at index J

SISAL - A[J]

IF1 -

```
          array(T)     integer
             |            |
             v            v
      ------------------------------
      |                            |
      |         AELEMENT           |
      |                            |
      ------------------------------
             |
             v
             T
```

Runtime - select

Params:

NAME	MEANING	SOURCE
dvptr	ptr to dope vector to start dereferencing at	from symbol table
rsize	element size	array type
ndim	number of levels in the array	array type
boundarr	ptr to subscripts to apply to array	input edges

Replacement - replaces the array element at index J with value V

SISAL - A[J:V]

IF1 -

```
      array(T)   int   value(s)
         |        |       |
         v        v       v
      ------------------------------
      |                            |
      |         AREPLACE           |
      |                            |
      ------------------------------
         |
         v
      array(T)
```

Runtime - replace

Params:

NAME	MEANING	SOURCE
dvptr	ptr to dope vector of input array	symbol table
newdvptr	new dope vector	output param
dvflag	boolean indicating if element to replace is an array	array type
numval	number of elements to replace	input edges

```
      rsize      element size               array type
      ndimP      if values are arrays,      array type
                 number of dimensions in
                 these arrays
      boundarr   array of indices to the    input edges
                 element
      valarr     array of ptrs to values    input edges
      inplace    boolean indicating if          ?
                 input array is referenced
                 again
```

Concatenation - concatenates two or more arrays

SISAL - A||B

IF1 -

```
                array(T)   array(T)
                   |          |
                   v          v
                ------------------------
                |                      |
                |     ACATENATE        |
                |                      |
                ------------------------
                   |
                   v
                array(T)
```

Runtime - concat

Params:

```
      dvflag     boolean indicating if      array type
                 input arrays contain
                 dope vectors
      numarr     number of arrays being     count input edges
                 concatenated
      rsize      element size               array type
      newdvptr   new dope vector            output param
      dvarr      array of ptrs to dope      input edges and
                 vectors for input arrays   symbol table
```

Add high/Add low - appends a single value to either end of an array

SISAL - array_addh(A,V)   array_addl(A,V)

IF1 -

```
         array(T)            value
            |                  |
            v                  v
         ----------------------------
         |                          |
         |       AADDL/AADDH        |
         |                          |
         ----------------------------
            |
            v
         array(T)
```

Runtime - [illegible]dh, [illegible]dl

Params:

NAME	MEANING	SOURCE
dvptr	ptr to dope vector	symbol table
newdvptr	new dope vector	output param
value	ptr to value to append	input edge
nsize	size of value	edge type
vflag	boolean indicating if value is an array	edge type

Remove high/low element - returns the array, A with its high index decreased by one or its low index increased by one

SISAL - array_remh(A)   array_reml(A)

IF1:

```
          array(T)
             !
             v
     ---------------
     !             !
     ! AREML/AREMH !
     !             !
     ---------------
             !
             v
          array(T)
```

Runtime - array_reml, array_remh

Params:

NAME	MEANING	SOURCE
dvptr	ptr to input array	symbol table
newdvptr	new dope vector	output param
nsize	element size of array	array type

Set low limit - adds LO - array_lim(A) to all elements thus shifting the origin of the array

SISAL - array_setl(A,LO)

IF1 -

```
     array(T)         LO
        !              !
        v              v
     ------------------
     !                !
     !     ASETL      !
     !                !
     ------------------
        !
        v
     array(T)
```

Runtime - array_setl

Params:

NAME	MEANING	SOURCE
dvptr	ptr to input array	symbol table
newdvptr	new dope vector	output param
lobound	lower bound for new array	input edge

Set bounds - returns an array with range (LO,HI) with the same data as the input array where possible. If LO > array_liml(A) or HI < array_limh(A) elements will be missing

SISAL - array_adjust(A,LO,HI)

IF1 -

```
          array(T)   LO    HI
              :       :     :
              v       v     v
        ----------------------------
        :                          :
        :        AADJUST           :
        :                          :
        ----------------------------
              :
              v
          array(T)
```

Runtime - array_adjust

Params:

NAME	MEANING	SOURCE
lo	lower bound	input edge
hi	upper bound	input edge
rsize	element size of array	array type
dvptr	pointer to input array	symbol table
newdvptr	new dope vector	output param

4.2 ANALYSIS RESULTS

There were several observations from the above analysis: 1) The majority of the input parameters needed by the runtime routines are directly available from either the IF1 or the symbol table. The only exception to this was parameter "rinplace" used in the replace routine to determine if the input array is ever used again. If it isn't, the replacement can take

place without having to copy the array. Some kind of "live" analysis needs to be done to supply this value.

2) Although both SISAL and the runtime routines allow for shortcuts in referencing a multi dimensional array, the IF1 breaks them down into indexing by only one index at a time. This is done to allow loop invariant removal and common subexpression optimizations to be performed, however, it means some of the capabilities of the runtime routine will never be used. The actions taken for one call to the runtime routine versus several calls to dereference an element are probably not much different, however a routine that would combine the references into one IF1 node after all optimizations have been performed could be useful.

3) A similar thing happens with concatenates as they are broken down into concatenating only two arrays at one time by the intermediate form. Both SISAL and the runtime routine will allow multiple arrays to be concatenated at once. In this case, however, the results of several calls to the runtime routine versus just one can result in less efficiency. This happens because for each call a new logical array will be allocated to hold the results of the concatenate. One call to the runtime routine would only have to allocate one logical array. An optimization that would go through the IF1 and combine any series of concatenates into one could increase the efficiency of the runtime routine.

4) There is an IF1 node available called AIsEmpty which returns a boolean indicating if there are any elements in the array. I couldn't find any SISAL statement that used this node.

5) There is a runtime routine which deallocates arrays. I was unable to find any documentation that outlined when an array would be deallocated.

From these observations it was decided that routines were needed to combine a series of concatenates into one concatenate or a series of selects into one select. A routine was also needed to do a "live" analysis on an array input to a replace node. Instead of writing a specific routine for this purpose, however, a generalized routine which given any input edge will determine if the value is ever needed again would satisfy this requirement and would also be useful in other areas of code generation.

4.3 COMBINING CONCATENATES

As mentioned above, a routine was written to combine a series of concatenate nodes in IF1 into a single node when possible. Figure 3 shows an example of an IF1 graph with several concatenate nodes and the graph that would result from this routine. The code for this routine is contained in appendix B and described below.

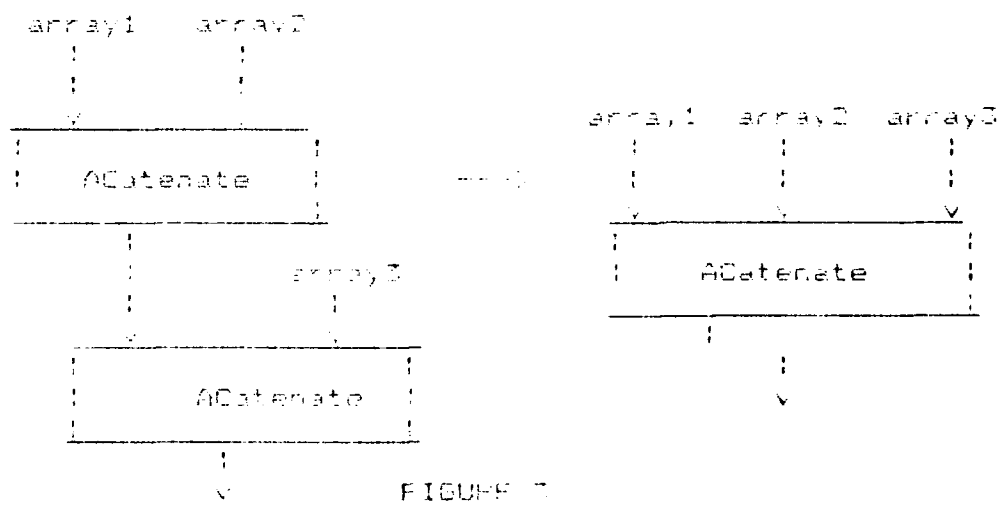


FIGURE 3

The main strategy is to start with a concatenate node from the IF1 graph and then by following its inputs recursively backwards through the graph to determine if any of the inputs to this node are also concatenates. If so, the input edges of the higher concatenate are linked into the lower concatenate node instead. Before the edges are linked to the lower node, however, its inputs are also checked so that multiple levels of concatenates can ultimately be combined together.

The main routine is called "combinecats" and expects as input a pointer to a concatenate node. It first initializes a linked list which will be used to hold the input edges to the final combined concatenate node. It then calls a routine "followedges" which will cycle back through all the input edges to the node and place the inputs to all nodes that can be combined in the linked list. After it returns from this routine it sets the backptr in the original node to point to this new list of input edges and sets the next pointer of the last edge to NULL.

Routine "followedges" cycles through each of the input edges to a node. For each edge it looks at the port that produced this value and checks if there is more than one use of this value. If so, it does not combine the nodes since this intermediate value is needed somewhere else. If the value is only used once it traces back to the source node. If it is a concatenate node, the nodes can be combined and it calls itself to start checking the inputs for this node. Once an input edge is found whose source node can not be combined the input edge is added to the linked list of input edges for the combined node.

In addition, the corresponding "forward" edge for this input value must be made to point to the combined node instead of the original node that may have already been combined. Once all edges for the input node have been processed the routine returns. Because of the order that edges will be processed in this routine the edges placed on the linked list of edges will end up in the correct order.

Figure 4a is a detailed example of the IF1 graphical encoding of two concatenates that can be combined into one. Combineconcats is called with a pointer to node 5. It in turn calls followedges with node 5. Followedges looks at the first input edge. By going back up to the source port and then looking at the linked list of edges pointed to by this port it sees that there is only one use of this value. It also sees that the source node is a concatenate and so it calls itself with a pointer to node 3. The first input to node 3 is checked and although there is only one use of this value the source node is not a concatenate. This means this input edge must be place on the linked list and the edge pointed to by node 1 port 1 must be made to point to node 5 instead of node 3. Now the second input to node 3 is checked. It is similar to edge 1 and so it is also added to the linked list and its corresponding forward edge made to point to node 5. This completes the processing of node 3. Next, the second input edge to node 5 is processed. This time it is found that the input array value is needed by another node and thus its source node can not be combined. The edge is therefore put on the linked list of edges and its forward edge made to point to node 5 (which it already did).

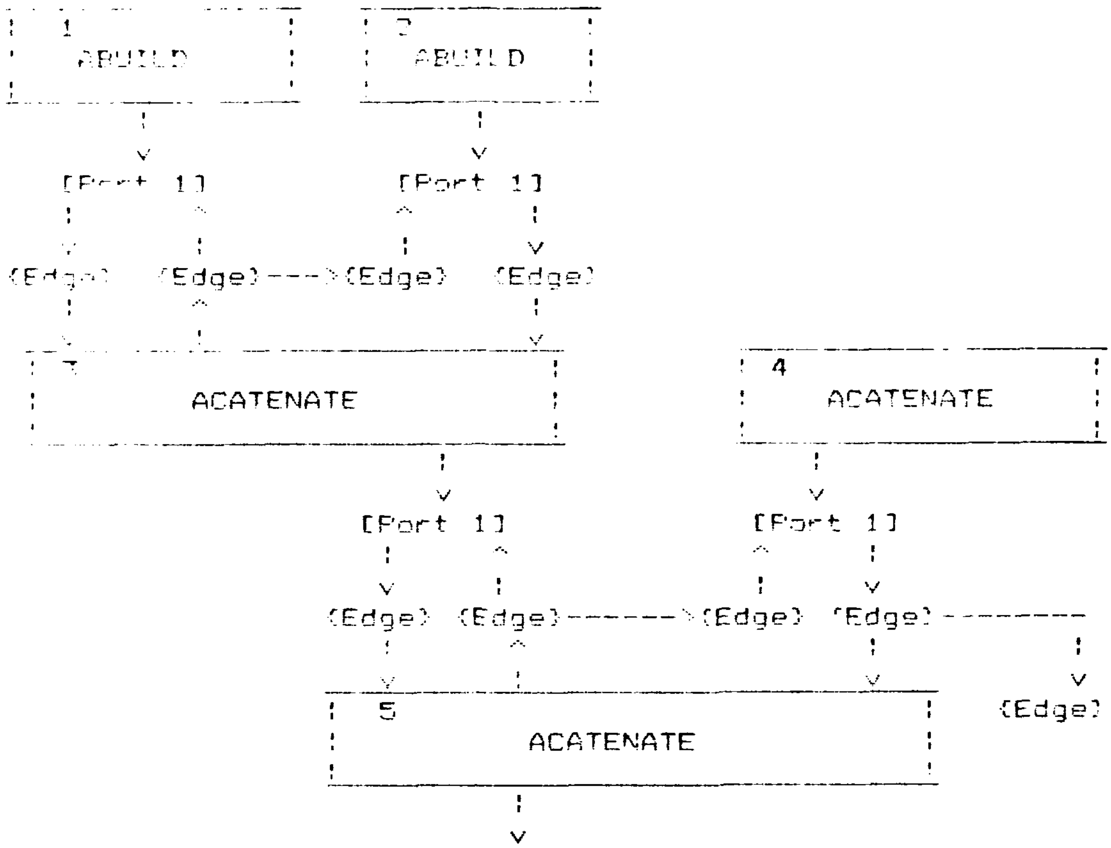


FIGURE 4a

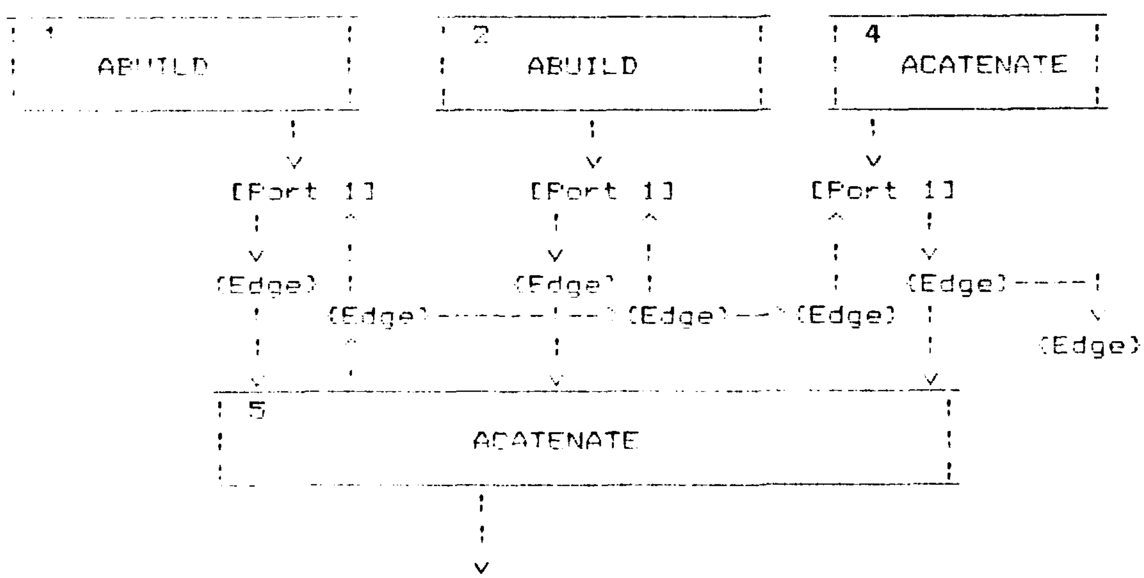


FIGURE 4b

This is the last input edge for node 5 and so "followedge" returns to "combinearrate". The backptr for node 5 is put to the linked list of edges and the next pointer for the last edge set to NULL. The resulting graph is shown in figure 4b.

4.1 COMBINING SELECTS

The routine to combine selects is very similar to that described above for concatenate above. The goal being to combine the dereferencing of several dimensions of a multi-dimensional array into one runtime call. The code for this routine is contained in appendix C and described below.

Again, the strategy is to start with a select node from the IF1 graph and then by following its inputs recursively backwards through the graph to determine if any of its inputs are also selects. If they are and there is no other use of the input value then the index for this dimension of the array is added to a list of indexes to be used in dereferncing an element.

The main routine is called "combinesel" and expects as input a pointer to a select or "aelement" node. It initializes a linked list which will hold all the indexes for the final select node. It then calls routine "followback" to follow through all the input edges for this node and to place all the indexes that can be combined in the linked list. After it returns from this routine it fixes up the pointers to the node so that its inputs are the array to be dereferenced followed by all the indexes. The array to be dereferenced is the first input to the highest node in the graph that could be combined.

Routine "followback" looks at the first input to the node which is the array that is input to the select. It looks at the source node for this input and if it is also a select and there is no other use of this intermediate value, then the nodes can be combined. It then calls itself with this new node to see if the input to this node can also be combined. Once a node that cannot be combined is found, the routine adds the index for the current node onto the linked list of indexes. The corresponding "forward" edge is also made to point to the combined node. As the routine returns from all its recursive calls, all indexes will be added to the linked list.

4.5 LIVE ANALYSIS

Given a pointer to an edge, this routine will return 0 if the value is not used again or 1 if it is. The basic strategy here is to determine if there are any other uses of the value produced by the source port of this edge. Because of the parallel manner in which the IF1 nodes are executed, any other use of a value could possibly be a later use of the value. The only exception would be if there was a data dependency that required the other use of this value to occur before the particular use being analyzed.

The code for live analysis is contained in Appendix D and described here. The main routine is called "islive" and expects as input a pointer to the edge whose "liveness" is in question and a pointer to the node that this edge is input to. The edge must be the "back edge" that one would find by following the backlinks from a node, not the "forward edge" one would get by

following the usage pointer from a node. The [illegible] initializes [illegible] to be false (0). It then finds the port that produced the value and makes a list of all other nodes that use this value. Associated with each node on the list is a flag which is initially set to 1 which means that the node is live. If there are any nodes on the list, (i.e. there are other uses), it calls "checkdepn" to see if there is a data dependency that would cause the node using the edge in question to occur after the other nodes that use this value. If any node on the list does not have such a dependency then the value is considered to be live. If no live use of this value has been found, the next step is to determine if the edge in question is imbedded in a compound node and was passed in from outside the node. If so it is possible that there is another use of the value outside of the compound node. The graph node for the compound node and the corresponding input edge is found. The liveness of the original edge is then determined by calling "islive" with this corresponding edge.

Routine "checkdepn" takes as input a pointer to a node that is being checked and a pointer to a list of nodes against which it is to be checked for data dependencies. The routine cycles through each input edge to the node to be checked and finds the [illegible] for this value. It then cycles through the linked list of nodes and sets its flag to 0 if a match is found. This means that there is a data dependency and the node is no longer live. If the node is not node 0 (the graph node) it recursively calls itself with this source node to continue checking for data dependencies.

[illegible] ARRAY OPTIMIZATIONS CONCLUSION

Th[illegible] the [illegible] discussed above will help improve the [illegible] of [illegible] used in SISAL. The [illegible] large [illegible] to be [illegible] all the [illegible] this analysis [illegible] would have to result in another copy of the array being [illegible]. This would get extremely costly.

There are two problems associated with the routines that [illegible] user should be aware of. The first is that the node resulting from combining several concatenate nodes or several selects is theoretically an illegal IF1 node since it has more than two inputs. It was believed that the only user of the IF1 graph after these optimizations were done would be the code generator and thus this would not be a problem. Any other user who might look at this graph at this point should be aware of these irregular nodes. The second problem is in the graphical encoding of IF1. Currently there is no way to jump back out of a compound node to the surrounding graph. This causes problems in the [illegible] analysis when one wants to jump out and check for other uses of this value. A pointer will have to be added to the graphical encoding to allow this.

5.2 STREAM OPTIMIZATIONS

The next objective of this project was to analyze the [illegible] operations in SISAL to determine if any optimizations [illegible] make them more efficient. Like arrays, this involved first analyzing the relationships between SISAL, IF1 and the corresponding stream runtime routines. Unfortunately, [illegible]

[illegible] documentation outlining the stream runtime routines [illegible] parameters they require. I [illegible] currently waiting for [illegible] of this information and thus the entire analysis could not be completed. The SISAL IF1 relationship was analyzed and is shown in the next section.

5.[illegible] STREAM ANALYSIS

Create a stream - creates a stream of the desired type with no elements

SISAL - stream (type-name)[]

IF1 - A stream type is created. No node is generated.

Append - add value V to the end of stream S

SISAL - stream_append(S,V)

IF1 -

```
                 stream(T)     value
                     !           !
                     v           v
                 ---------------------
                 !                   !
                 !       AADDH       !
                 !                   !
                 ---------------------
                     !
                     v
                 stream(T)
```

Runtime -

Select first element - returns the first element of a stream

SISAL - stream_first(S)

IF1 -

```
                 stream(T)      1
                     !          !
                 ----v----------v---
                 !                 !
                 !    AELEMENT     !
                 !                 !
                 -------------------
                     !
                     v
                     T
```

Runtime -

Select all but first element - returns a stream identical to the input stream except with the first element removed

SISAL - stream_rest(G)

IF1 -

```
          stream(T)
              |
              v
       ---------------
       |             |
       |    AREML    |
       |             |
       ---------------
              |
              v
          stream(T)
```

Runtime -

Test for empty - returns true if there are no elements in the stream and false otherwise

SISAL - stream_empty(G)

IF1 -

```
          stream(T)
              |
              v
       -------------
       |           |
       | AISEMPTY  |
       |           |
       -------------
              |
              v
          boolean
```

Runtime -

Number of elements - returns the number of elements in G

SISAL - stream_size(G)

IF1 -

```
                 stream(G)
                     |
                     v
              ----------------
              |              |
              |    ASIZE     |
              |              |
              ----------------
                     |
                     v
                  integer
```

Runtime -

Concatenate - returns a stream of all elements of G followed by all elements of H

SISAL - G || H

IF1 -

```
        stream(T)      stream(T)
            |              |
            v              v
        ---------------------------
        |                         |
        |       ACATENATE         |
        |                         |
        ---------------------------
            |
            v
        stream(T)
```

Runtime -

5.2 STREAM ANALYSIS CONCLUSIONS

Not much can be said until the runtime routines are analyzed. It seems though that the same sort of things done for arrays may need to be done with streams. In particular, concatenates could possibly be combined if the runtime routine can handle more that two at a time. Also some kind of live analysis may be useful in determining when a stream is no longer needed.

6.0 PROJECT CONCLUSIONS

The goal of this project was to provide some optimizations for the SISAL compiler which would allow it to produce more efficient code. Since SISAL is compiled into the intermediate form IF1 which is a common intermediate form for data flow languages we were looking for optimizations that were unique to the characteristics of SISAL rather than the more traditional optimizations. As we found out with loop optimization, these more traditional optimizations are being worked on at other sites. Arrays were one area of SISAL that could be a potential problem area if no optimizations were applied. In particular, replace operations could quickly eat up a lot of memory if the array had to be copied each time. The combining of concatenate and select nodes and the live analysis provided by this project will hopefully increase the efficiency of array operations in SISAL.

As for my personal gains from this project, I learned a lot about data flow machines and date flow programming languages. The concepts of data flow and parallel execution add another dimension to the way one traditionally thinks about programming. Looking at the loop optimizations also forced me to analyze those methods more thoroughly.

REFERENCES

[Cobb,1984] Stephen F. Cobb, et. al. Arrays in SISAL. Computer Science Technical Report CS-84-04, Colorado State University, July 1984.

[McGraw,1983] James McGraw, et. al. SISAL: Streams and Iteration in a Single-Assignment Language, Language Reference Manual. Version 1.1, July 20, 1983.

[Skedzielewski,1984] S. Skedzielewski. IF1 - An Intermediate Form for Applicative Languages. Draft 9, 1984.

APPENDIX A

IF1loop Documentation

IF1loop is a command that can be executed to perform loop optimizations on an IF1 file. This documentation describes the routines that perform this optimization. The main routine is [illegible]Improver.

AddInputPorts

This function will add input ports to a loop graph. This involves adding ports to the graph as well as its subgraphs (i.e. the test, body, and results subgraphs).

Inputs   Pointer to the node to add ports to
         Number of ports to add
Output   None
Called by   MoveNodeOut
Calls   None

CompactPorts

This routine cycles through the input ports for a loop node and deletes any that are no longer used.

Inputs - Pointer to node whose ports will be compacted
Output - None
Called by   ImproveCompoundNode
Calls   None

ImproveCompoundNode

This function cycles through the nodes of a loop and call IsMoveable to determine if a node is invariant. If so, it call MoveNodeOut to move the node into the graph surrounding the loop. After processing all nodes in the loop, if any nodes were moved it calls CompactPorts to renumber the inputs to the loop to reflect deletions and additions of inputs caused by the moves.

Inputs - Pointer to the compound node to be checked for invariants
Output - None
Called by - ImproveGraph
Calls   IsMoveable
            MoveNodeOut
            CompactPorts

ImproveGraph -

Cycles through the graph of a function looking form compound nodes. When one is found it calls itself to improve the subgraphs of this graph. It then checks whether the node is a loop construct and call ImproveCompoundNode to remove the loop invariants. After the entire graph has been processed it call RemoveGraphCSE to remove common subexpressions.

Inputs - Pointer to the first node in graph
Output - None
Called by - LoopImprover
Calls - ImproveGraph
ImproveCompoundNode
RemoveGraphCSE

IsMoveable -

This function determines whether a node can be moved. A node can be moved if: a) all inputs are loop constants or b) it is not associated with manipulating multiple values.

Inputs - Pointer to the node to be checked
Pointer to the surrounding graph node
Number of current inputs to the surrounding graph
Output - Boolean indicating if a node can be moved
Called by - ImproveCompoundNode
Calls - None

LoopImprover -

This is the main procedure for the loop optimizations. It cycles through the functions of a program calling other routines to perform the optimizations on each function.

Inputs - Pointer to a function in the program
Outputs - Indirectly an improved IF1 graph
Called by - None
Calls - ImproveGraph
RenumberGraph

MoveNodeOut -

This function moves a node from one location to another by first changing the input edges to the node to reflect the environment of the outer graph. Next, AddInputPorts is called to create new input ports into the loop to hold the outputs of those moved nodes and their outputs are wired into these ports.

Inputs - Pointer to the node to be moved
Pointer to subgraph in compound node that the node to

be moved resides in
Pointer to compound node that the node is to be moved
out of
Output - Current number of input ports to the compound node
Called by - ImproveCompoundNode
Calls - AddInputPorts

APPENDIX B

CODE TO COMBINE CONCATENATE NODES

```
#include "siatl/if.encode/symdef.h"
struct edgetype *eliststart, *elistend;
      /* Used to keep a linked list of input edges for the */
      /* combined node */
struct IFnode *masternode;
      /* Points to the combined node */


/**************************************************************/
/*                     COMBINECATS                            */
/*  This routine will combine a series of concatenate nodes*/
/*  by following the input edges of a concatenate node and */
/*  and linking the input edges of all nodes that can be   */
/*  combined into one node.                                */
/**************************************************************/

combinecats(nodeptr)
    struct IFnode *nodeptr;       /* Points to node to start */
                                  /* combining at */
{
    masternode = nodeptr;        /* Remember this starting node */
    eliststart = elistend = NULL; /*Initialize list of edges  */
    edgeno = 1;                  /* Used to renumber edges */
    followedges(nodeptr);        /* Call followedge to do the   */
                                 /* actual combining */
    nodeptr->backptr = eliststart; /* Make node point to this */
                                 /* new list of edges */
    elistend->nextedge = NULL; /* fix last edge on list */
}


/**************************************************************/
/*                     FOLLOWEDGES                            */
/*  This routine will cycle recursively backwards through  */
/*  a graph determining if nodes can be combined.  When it */
/*  reaches the last node that can be combined on a given  */
/*  path it puts its input edges on a list of edges for the */
/*  final combined node.                                   */
/**************************************************************/

followedges(nodeptr)
struct  IFnode *nodeptr; /* points to the node whose edges */
                          /* are to be followed */
{
    struct edgetype *sourceedge, *destedge;
    struct porttype *sourceport;

    sourceedge  = nodeptr->backptr;  /* get first input edge */
/* now cycle through all its input edges checking if the    */
```

```
/* input's source node can be combined */

    while (sourcedge != NULL) {
        sourceport = sourcedge->ptr.up;
    /* if this value is not needed elsewhere and the source */
    /* node is a concstenate it can be combined */
        if ((sourceport->usage->nextedge == NULL) &&
            (sourceport->source->typenode == catenate)) {
            sourceport->source->beenhere = 1;
            followedges(sourceport->source);
        }
        else  {    /* the source node can not be comined and */
            /* this edge is put on the edge list */
            if (eliststart == NULL)
                eliststart = sourcedge;
            else
                elistend->nextedge = sourcedge;
            elistend = sourcedge;
        /* the corresponding "forward edge" must be made to */
        /* point to the final combined node */
            destedge = sourcedge->ptr.up->usage;
            while (destedge->ptr.dn != nodeptr)
                destedge = destedge->nextedge;
            destedge->ptr.dn = masternode;
            destedge->port = sourcedge->port = edgeno++;
        }
        sourcedge = sourcedge->nextedge;
    }
}
```

APPENDIX C

CODE TO COMBINE SELECT NODES

```
#include "sisal/if.encode/symdef.h"
struct edgetype *eliststart, *elistend;
       /* Used to keep a linked list of input edges for the */
        * combined node */
struct IFnode *masternode;
       /* Points to the combined node */
struct IFnode *topnode;
       /*Points to the highest node in the graph */
       /* to be combined */

/***********************************************************/
/*                       COMBINESEL                        */
/*  This routine will combine a series of select or        */
/*  "element" nodes by following the array input to the    */
/*  node and linking the indexes of all nodes that can be  */
 *  combined into one node.                                */
/***********************************************************/

combinesel(nodeptr)
    struct IFnode *nodeptr;         /* Points to node to start */
                                    /* combining at */
{
    masternode = nodeptr;           /* Remember this starting node */
    topnode = nodeptr;              /* initialize the highest node */
                                    /* to be the current node */
    eliststart = elistend = NULL; /*Initialize list of indexes*/
    edgenum = 0;                    /* Used to renumber edges */
    followback(nodeptr);            /* Call followback to do the   */
                                    /* actual combining */
    nodeptr->backptr = topnode->backptr; /* make node point to*/
                                    /* input array */
    nodeptr->backptr->nextedge = eliststart; /*link in list of*/
                                    /* indexes */
    elistend->nextedge = NULL; /* fix last edge on list */
    /* now fix up first input edge */
    edgeptr = topnode->backptr->ptr.up->usage;
    while (edgeptr->ptr.dn != topnode)
         edgeptr = edgeptr->nextedge;
    edgeptr->ptr.dn = nodeptr;
    edgeptr->port = 1;
}
```

```
/*****************************************************************/
/*                        FOLLOWBACK                             */
/*  This routine will cycle recursively backwards through        */
/* a graph determining if nodes can be combined.  When it        */
/* reaches the last node that can be combined on a given         */
/* path it puts its index on a list of indexes for the           */
/* final combined node.                                          */
/*****************************************************************/

followback(nodeptr)
struct  IFnode *nodeptr; /* points to the node whose edges */
                         /* are to be followed */
{

    struct edgetype *sourcedge, *destedge;
    struct porttype *sourceport;

    /* now check if the input's source node can be combined */

    sourceport = nodeptr->backptr->prt.up;
    /* if this value is not needed elsewhere and the source */
    /* node is an element it can be combined */
    if ((sourceport->usage->nextedge == NULL) &&
            (sourceport->source->typenode == arrayelement)) {
            topnode = sourceport->source;
            topnode->beenhere = 1;
            followback(topnode);
        }
        sourcedge = nodeptr->backptr->nextedge;
        /* put the index on the linked list */
        if (eliststart == NULL)
            eliststart = sourcedge;
        else
            elistend->nextedge = sourcedge;
        elistend = sourcedge;
        /* the corresponding "forward edge" must be made to */
        /* point to the final combined node */
        destedge = sourcedge->ptr.up->usage;
        while (destedge->ptr.dn != nodeptr)
            destedge = destedge->nextedge;
        destedge->ptr.dn = masternode;
        destedge->port = sourcedge->port = edgeno++;
        }
        sourcedge = sourcedge->nextedge;
    }
}
```

APPENDIX D

CODE TO TEST IF AN EDGE IS LIVE

```
#include "sisal/if.encode/symdef.h"
struct nodelist {
    struct IFnode *nodeptr;
    int depends;
    struct nodelist *nextnode;
}; /* This structure will be used to hold a list nodes that */
 /* also use the value being tested */


/*****************************************************************/
/*                        ISLIVE                                 */
/*    This routine checks if a given data value  represented     */
/* by an edge will ever be used again.                           */
/*****************************************************************/

islive(innode,inedge)
    struct IFnode *innode      /* points to the node that the */
                               /* edge in question is input to*/
    struct edgetype *inedge;   /* points to the "backedge" */
                               /* whose liveness is to be  */
                               /* tested */
{
    struct nodelist *nliststart, *nptr; /* contains a linked */
                              /* list of other nodes that use  */
                              /* this data value */
    struct edgetype *edgeptr;
    int live, found;

    live = 0;         /* initialize live to false */

    /* build list of all other uses */

    nliststart = NULL;
    edgeptr = inedge->ptr.up->usage;
    while (edgeptr != NULL) {
        if ((edgeptr->ptr.dn != innode) ||
            (edgeptr->port != inedge->port)) {
            nptr   =   (struct  nodelist  *)(malloc(sizeof
                          (struct nodelist)));
            nptr->nodeptr = edgeptr->ptr.dn;
            nptr->depends = 1; /* initialize it to be live */
            nptr->nextnode = nliststart;
            nliststart = nptr;
        };
        edgeptr = edgeptr->nextedge;
    };
```

```
	/* [illegible] all inputs to this node to see if they */
	/* [illegible] there [illegible] */

	[illegible] (inlist != NULL) {
		[illegible] (node, inlist);
		[illegible] = [illegible];
		/* cycle through inset and see if any are still live */
		while (nptr != NULL) {
			live = live || nptr->depends;
			nptr = nptr->next_node;
		}
	};

/* [illegible] edge [illegible] live, check if it is inside a */
/* compound node. If so, and this value was passed into */
/* [illegible] from outside, find its corresponding edge and */
/* [illegible] if live */

	if ([illegible] inedge->ptr.up->source->nodenum == 0) && (!live) &&
		[illegible] value.graphnode.compnode
		[illegible] {
		[illegible] inedge->ptr.up->source->value.graphnode.compnode;
		[illegible] ptr;
		found = 0;
		/* find the corresponding edge outside of the compound*/
		/* node */
		while (!found) {
			if (edgeptr->port == inedge->ptr.up->port)
				found = 1;
			else
				edgeptr = edgeptr->ne_tedge;
		}
		live = isalive(edgeptr);
	}

	return (live);
}

/*********************************************************************
*                           DEPCHECK                                 */
/*  This routine will check if a node has any data                   */
/*  dependencies that would [illegible] execute after other          */
/* nodes. This is done by calling [illegible] through                */
/* its input edges to see if they depend on the output of            */
/* the other nodes.
*********************************************************************/

[illegible] depcheck[illegible])
	[illegible] /* [illegible] to [illegible] to be checked */
				/* for data dependencies */
	[illegible] /* pointer to a list */
				/* of nodes to check the node above */
				/* against */

	[illegible] *node [illegible];
```

```
[illegible] edgeptr;
[illegible]

edgeptr = [illegible]->inlistptr; /* get pointer to first input */
                                  /* edge for the node */
/* follow the path of each input edge checking if it */
/* depends on the [illegible] of the nodes on the list */
while (edgeptr != NULL) {
        sourcenode = edgeptr->ptr.op->source[illegible];
    if (sourcenode != NULL) {
        nptr = nliststart;
        while (nptr != NULL) {
            if (sourcenode == nptr->nodeptr)
                nptr->depends = 1;
            nptr = nptr->nextnode;
        }
        /* if we haven't hit the graph node keep  */
        /* recursively checking the inputs for    */
        /* this node */
        if (sourcenode->nodenum != 0)
            checkdepn(sourcenode, nliststart);
    }
    edgeptr = edgeptr->nextedge;
  }
}
```